ARM Assembly for Embedded Applications, 5th edition
ISBN 978-1092542234

CHAPTER 3: WRITING FUNCTIONS IN ASSEMBLY

Section 3.4 (Function Return Value), page 45, Figure 3-9, right-hand column:

Replace next to last instruction LDR R2,=save8 with LDR R2,=save64

Added new section (subsequent sections renumbered):

(Corresponding changes made to Figures 3-12 and 3-13 as mentioned in the new section.)

3.6 STACK ALIGNMENT

Some library functions and functions compiled from a high-level language are written in such a manner that requires the address held in the stack pointer to be a multiple of eight when the function is called. The compiler ensures that the stack pointer is always a multiple of eight whenever your assembly language function is called from a high-level language program. However, the value of the stack pointer will no longer be a multiple of eight if your function then calls another function after pushing an odd number of registers. Thus, when writing a function in assembly that calls a library function or a function that was compiled from a high-level language, always push and pop an *even* number of registers. As shown in Figure 3-12 and Figure 3-13, this is accomplished by selecting register R3 to add to the PUSH and POP list since there is no other reason to do so.

CHAPTER 4: COPYING DATA

Section 4.6 (Examples of Copying Data), page 67, Figure 4-12:

The data type names in the figure are incorrect. E.g., "`int_8`" should be "`int8_t`", etc.

Section 4.11 (Copying a Block of Data Quickly), page 79, Table 4-7, "Operation" column:

The italicized (1st) line of the 2nd row should be: *Rn -= 4 × #regs; registers ← memory*
The italicized (1st) line of the 3rd row should be: *registers → memory; Rn += 4 × #regs*

CHAPTER 6: MAKING DECISIONS AND WRITING LOOP

Section 6.4 (Comparing 64-bit Integers), page 110:

Replace the last paragraph and subsequent code by the following:

> Since a compare is simply a subtraction that discards the difference but records the characteristics of that result in the flags, one might assume that all that is needed is to simply perform a 64-bit subtraction and check the resulting flags. However, although the `N`, `V` and `C` flags will be correct, the `Z` flag may not since it will only indicate if the most-significant half of the difference is zero.
>
> When the condition to be tested does *not* depend on the `Z` flag (`GE,LT,HS,LO,MI,PL,VS,VC,AL`), then no correction is necessary and the condition test may immediately follow the 64-bit subtraction:


```
...                     // Load operands as before
SUBS      R0,R0,R2      // subtract LS halves, capture borrow
SBCS      R1,R1,R3      // subtract MS halves w/brw; set flags
...                     // OK to test GE,LT,HS,LO,MI,PL,VS,VC,AL
```

However, all other conditions (`EQ,NE,GT,LE,HI,LS`) require a different approach. The solution for `EQ` and `NE` is quite simple:

```
...                     // Load operands as before
SUBS      R0,R0,R2      // compute LS half of difference
SBC       R1,R1,R3      // compute MS half of difference
ORRS      R1,R0,R1      // Z=1 iff both halves are zero
...                     // Now OK to test for EQ or NE
```

For `LE`, `GT`, `LS` and `HI` we can avoid testing the `Z` flag by reversing the operands. Since `x <= y` is equivalent to `y >= x`, this allows us to replace `LE` by `GE` or `LS` by `HS`, which don't require testing the value of `Z`:

```
...                     // Load operands as before
SUBS      R2,R2,R0      // subtract LS halves (operands reversed)
SBCS      R3,R3,R1      // subtract MS halves (operands reversed)
...                     // Replace LE/GT by GE/LT, or
...                     // Replace LS/HI by HS/LO.
```

Finally, note that in all cases except `EQ` and `NE`, we can avoid modifying one register by replacing the `SUBS` instruction by a `CMP` instruction.

CHAPTER 8: MULTIPLICATION AND DIVISION REVISTED

Section 8.3 (Division by an Arbitrary Constant), page 171:

Replace the code at the top of the page by:

```
// Divide by +7 (-7: replace LSRS.N, ADD by ASRS.N, SUB)

LDR     R0,=dividend
LDR     R0,[R0]
LDR     R1,=2454267027    // 2 clock cycles
SMULL   R2,R1,R1,R0       // 1 clock cycle   } Use SMMLA
ADDS.N  R1,R1,R0          // 1 clock cycle   } (see below)
LSRS.N  R0,R0,31          // 1 clock cycle
ADD     R0,R0,R1,ASR 2    // 1 clock cycle
LDR     R1,=quotient
STR     R0,[R1]
```

Section 8.3 (Division by an Arbitrary Constant), page 171, paragraph that starts at the bottom of the page:

Replace the entire paragraph by the following:

> The SMMUL instruction computes the most-significant half of a 64-bit signed product and eliminates the need to use register R2 in the previous code. The SMMLA instruction does the same, but also adds the value of another 32-bit operand to the result. This instruction can thus replace the SMULL/ADDS.N sequence in the earlier code that divides by +7. The SMMLS instruction computes the difference of a 32-bit operand less the most-significant half of a 64-bit operand. Unfortunately, this doesn't help simplify the code that divides by -7. There are no unsigned versions of these instructions.

Section 8.3 (Division by an Arbitrary Constant), page 172, Table 8-6:

In the third column of the table, second row, replace "$Rd \leftarrow (Rn \times Rm)$ <63..32> $+ R_a$" by "$Rd \leftarrow R_a + (Rn \times Rm)$ <63..32>".

In the third column of the table, third row, replace "$Rd \leftarrow (Rn \times Rm)$ <63..32> $- R_a$" by "$Rd \leftarrow R_a - (Rn \times Rm)$ <63..32>".

CHAPTER 11: FIXED-POINT REALS

Section 11.5.6 (Multiplying a Q32 Value by a Fractional Value), page 250, Figure 11-7:

The wrong partial products are indicated as being equal to zero ("= 0"). Those that are zero are only those that are cross-hatched and have a large "X" on top of them.